11 Language Syntax and Elements of Language

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Basic syntax structure to create simple Java programs.
- Primitive data types and how they can be employed in the Java syntax.
- Keywords which are specified in Java and cannot be used to denote the names of methods and data instances.
- String options, which play a vital role in the syntax of Java and how we can use its class structure.
- Various ways in which programmers can use classes, objects, and methods.
- Concept of implementing logic in Java programs by creating relevant classes and following the ideal OOP principles.
- Wrapper classes, autoboxing, unboxing, etc.
- Various operators.

In Chapter 10, you were introduced to some of the concepts of Java. We only tried to give you some basic idea of Java, program structure, syntax, etc. In this chapter, we will explain these concepts more thoroughly along with examples.

Syntax is a linguistic term, which defines the set of rules that govern how a language can be employed. It includes the order of words, their structure, and suggests how they deliver information to the language users. If we think about programming languages, their syntax consists of the basic rules that govern the program development in their specific environments.

The goal of this chapter is to familiarize you with the Java language syntax. We will describe the various components that make up this language and suggest the important elements that you need to learn to become a Java programmer.

11.1 | Building Blocks of Java

1

Java is an object-oriented programming (OOP) language. It is a high-level language, which encourages developers to use high-end logic and focus on creating complex functions, rather than limiting their focus on understanding and developing the basic language tools for their programs. As we have defined syntax in the linguistics, the Java syntax is similarly a set of rules that define how to write code for the Java virtual machine (JVM) compilation.

Java derives its basic structure and element names from C and C++ programming languages. Although it has a different working structure from these languages, because of the absence of global elements, it still uses the ease of common English words that represent functionality on offer. The language employs the structure of classes.

All Java code is defined in terms of classes, which may often form part of a library. All instances of these classes are termed as *objects*. Java is slightly different from a pure OOP language since it has primitive data types that are present within the language to ensure improved performance. There are some features which are possible in Java but are restricted to ensure that Java programmers cannot make mistakes. Such mistakes are difficult to identify and resolve, especially in large and complex programs.

Here, we first represent a basic Java code element and then describe how it is structured to give an idea of the syntax of this programming language:

```
public class MyVeryFirstJavaProgram {
public static void main(String[]args) {
    // This code will present the message within the quotations ahead in the terminal
    display window.
    System.out.println("First Java Program");
}
}
```

This is a simple example. The first word is "public" is a keyword in Java. These are words which have special meaning. The second word is another special word of "class". Remember, all Java programs consist of classes and class objects. Think of a class, as a standalone definition, which provides a structure but does not have a body of its own. The class name here is `MyVeryFirstJavaProgram`, which is termed as *identifier*. An identifier is a unique word that points to a particular class, object, or a particular method.

The keyword "static" describes a specific meaning in the syntax. Then we find the `main()` method. This is a method which must be present in your Java program for it to actually execute something. This is always run first by the program and can contain various program objects, which may be described with the use of different classes and the creation of a hierarchy.

There are also some words in parentheses. These are words that work as arguments to the methods, which are employed in Java programs. The line that is inserted with two forward slash symbols is the comments line, where we write what this code section does. Comments are extremely useful, and programmers employ them all the time to gain useful information about various code elements. This is especially true when multiple people work on a single Java project, where it is important to know what each program section does to avoid problems.

The next line presents the use of a method. Remember, methods in Java are code elements that perform the actual tasks. They represent the implementation of various schemes and are often used from the inherent Java libraries which become available with Java Runtime Environment (JRE), which is installed on all systems that are programmed to run Java programs using JVM instances.

Now, we look at how the code works. System is a Java class that contains various objects and methods that offer console control to Java programmers. The method of `println()` prints the string message contained within the quotations and then returns the console cursor to a new line. This is an improvement over the traditional console outputs of other languages, where you must manually program a movement to the new line.

Remember, method instances are finished using the semicolon sign ";". Another way to create the structure of any method, class object, or other values in Java is to employ curly braces "{}". They are used to employ compounded statements, where you want multiple independent code lines as a single coding structure.

These are some basic elements that formulate a Java structure. Remember, Java recognizes the end of a statement with a semicolon. We only use spaces and moving to a new line as tools to help us organize the programming project in a visual manner. Program comments are also not a necessity for use in Java. However, we use them to create an effective programming code where any programmer can perform the required analysis and study the code to make further additions or required improvements.

You can write the program that we have described in any text editor environment. Although there are some excellent Java programming platforms like Maven, all programs can be created using any ASCII-based editor. This program can only be compiled when you save the program with an extension of **.java**. Remember, your filename should be the name of your main program class. In this case, our program will be named "MyVeryFirstJavaProgram.java", where it is perfectly ready to be compiled using a Java installation.

The Java compiler can then be run on the final program, where it will create a compiled file if the code elements are correct and do not produce a compiler error. This file is then prepared to be executed if a system has a Java environment installed on it. This process describes the entire structure of a Java program and how it achieves the required maturity to perform the intended functionality on a computer system.

11.1.1 Keywords

A keyword in a programming language is a reserved word that can only be employed in writing code to represent a specific scenario. There are about 50 keywords in Java. They have their specific meaning and represent special functions. All integrated development environments (IDEs) use a special highlighting scheme to mark such words. This makes it easier to read the Java syntax and find out how the code is arranged and supposed to function.

These keywords often function as providing the additional performance over the OOP potential. They especially include access modifiers and specific words that explain the basic functionality and describe how class objects will function. They also include other tasks such as memory handling and the intrinsic options which are available in the programming language. We have already covered this section with the list of keywords in Section 10.4.7 of Chapter 10.

Flash ? Quiz Can you use these keywords as variable names? If yes, give an example. If not, can you use these keywords with case change? For example, class is a keyword so you can instead use class as variable name.

11.1.2 Primitive Classes

As we have already discussed, there is still a need to include basic values, even in OOP languages. There are eight primitive classes, or ones that are often termed as *primitive wrapper classes*, which provide an identity to the common types of data that we must employ. Boolean represents a class, which describes the results of a decision-making element. The types of byte, short, int, and long describe integers. The types of float and double represent accurate decimal numbers. The char type represents a single character value. We will be covering this in more details in Chapter 17. For now, let us try to understand this concept. Listed below are the various data types.

- **boolean:** It describes two values of true and false, with the default option set at false. It can be used to record the results of decision-making.
- **byte:** An 8-bit signed integer data type which is described as 2's complement. It has allowed values from –128 to 127 and is excellent for use in limiting data storage in large arrays. It has a default value of 0.
- **char:** It is for recording Unicode characters that are stored as 16-bit single character. Its values start from *u0000* to *uffff*, which translate as 0 to 65,535 in numbers. The default value is set to \u0000.
- **short:** It is a data type that allows for 16-bit storage of signed numbers that range from –32768 to 32767. It offers greater range than a byte and has the same purpose of allowing the use of reduced memory in large arrays. The default value is 0.
- **int:** This is the primary integer data type that uses 32-bits, which originally allowed for storing signed integers ranging from –231 to 231–1. However, Java 8 and later, it allows the use of unsigned integers, which gives the range as 0 to 232–1. It stores numbers using the form of 2's complement. It is the primary integer primitive data type commonly employed in Java programs. The default value is 0.
- **long:** Also representing integers, it uses 64-bits where the numbers are stored using the 2's complement form, a convenient method for storing integers. The signed digits range from –263 to 263–1, while the unsigned integers present from Java 8 onwards can have values from 0 to 264–1. The default value is 0.
- **float:** This data type represents a floating-point number with single precision using IEEE 754 32-bit standard. The float is often used to represent floating numbers that not require a high degree of precisions. It is excellent or storing data in arrays. The default value remains 0.0f.
- **double:** This is a double precision floating point value storage data type. It uses 64-bit storage and offers a much higher range of accuracy, which is perfect for use in applications such as currency conversions. According to the coding use, you should always double to store your precision numbers.

Table 11.1 compares the values for the primitives.

Table 11.1 Primitive values comparison

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2^{7}	$2^{7}-1$
short	16	2	-2^{15}	$2^{15}-1$
int	32	4	-2^{31}	$2^{31}-1$
long	64	8	-2^{63}	$2^{63}-1$
float	32	4	1.4E-45	3.4028235E38
double	64	8	4.9E-324	1.7976931348623157E308

Any value which is stored using these primitive data types does not have to be defined in a class and is termed as *literal* in Java.

Flash ? Quiz Can int hold a value of 10.3?

11.1.3 Literals

Although Java is an OOP language, it still provides specific functionality by providing data literals. These are intrinsic data types that are simply too important to include in any programming language. They represent the specific instances of the primitive data types, including options such as numeric, Boolean, character, and string data representations. You express particular data value cases using these literals.

These literals allow us to represent various number schemes such as binary, octal, hexadecimal, and decimal notations. It also allows the use of floating point data instances while two Boolean values of *true* and *false* are also available. There are string literals that are represented by writing within a set of double quotations. The language also provides several character escape sequences that are triggered with the use of a backslash "\".

You can view the entire list for Java 11 by going through this related Oracle page: https://docs.oracle.com/javase/specs/jls/se11/html/jls-3.html#jls-3.10

11.1.4 Variables

Variables identify specific literal values by using unique names that have already been explained. The first word is a keyword, which describes the type of variable such as int, float, and String. The unique name follows. You can initialize the variable with a value when it is declared or simply leave the declaration and it is initialized later. The following are a few examples of variable instances:

1. `int num;`
2. `num=0;`
3. `int num2=5;`
4. `int num1=2, num4=6;`

These examples show how it is possible to use the variables in different ways. You can define multiple variables in a single notation, by separating each instance using the delimiter of a comma ",".

11.1.5 Code Blocks

As already discussed, we use curly braces to create blocks of code. These blocks act as singular elements to their attached objects, methods, and any other units of coding in Java. You will find that each program has several instances of these code blocks which represent some specific functionality. Given below is an example showing the use of various blocks:.

```
void new Method() {
    int firstnum;
    {
        int secondnum;
        firstnum=1;
        System.out.println("first number is "+firstnum);
    }
    firstnum=2;
    System.out.println("first number is now "+firstnum);
    secondnum=5;
    // This is an illegal use of variable identity, since a variable declared in a code
    block will not remain available as a universal variable outside of that particular
    code block
}
```

This is an excellent code block example, as it shows that we can use multiple nested code blocks. However, they all represent the quality of presenting local and global values. The local variable of second num will not work outside of its code block. However, if we create further nested loops within its block, it will continue to work with the branched blocks in a normal way.

QUICK CHALLENGE Think of a use case in which you will utilize the code blocks.

11.1.6 Comments

We have seen the additional elements in Java other than code. They are the comments that provide guidance to other programmers as well as yourself, so that you can easily review your code when you want. You can use traditional comments with which you may be familiar from other languages such as C and C++. These are comments that start with "/*" and end with "*/".

You can employ them in any number of lines, as moving to new line is not recognized by the compiler. You can also use the end of the line comments, which are initiated with the use of "//" symbol (just as in the example given above). This comment continues till the line is eliminated by moving to the next line. They are useful for providing the exact functionality of the particular code elements, and is a tool often employed in good program coding practices.

There is also a third way of using comments. These are termed as *documentation comments*, as they are used to add information above class, methods on the code lines which can be read as documentation for the code. The *javadoc* (https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html) document generator reads these comments and creates the necessary records. Given below is an instance of the use of these comments:

```
/**
*This is first line of documentation
*This is second line.
……
*The nth required line.
*/
```

This represents a block of documentation which is perfect for inserting in specific sections of your program to define its logic and functional use for other programmers.

11.2 | Calling the Main Method

A key part of the Java syntax structure is the main method. All applications prepared in Java must have an entry point to start the program in a physical environment. This is regardless of whether the application runs in the background on a console setting or provides an interactive graphical user interface (GUI). This is achieved by calling the `main()` method in Java.

It is a static method. We will discuss more about such methods in Section 11.2.1. However, what you must know about the main method is that it can hold multiple classes as they may be required for Java application. However, there is always the ideal use of an external class, which must only hold the main method and provide the instance for naming the Java file. This method in Java does not return a value and has the intrinsic design of always employing "void" to describe its return type.

11.2.1 Static and Non-Static Methods

All methods that are employed in Java always lie under the hierarchal structures of classes and objects. However, they can all be divided into two basic types:

1. **Static method:** This method is defined with the declaration of a class and is therefore available for use throughout any class instances. It may also be employed by mentioning the class followed by a dot to use the method infrastructure in a Java program. Static methods are usually publicly available for use throughout the program.

However, this situation creates a problem because only a single instance of these methods is possible. You cannot use them with multiple copies. Take the example of the `main()` method, which we know will only have a single instance. This makes it a suitable static method in Java programs. Remember, static methods can be used by classes and by any object of the class. However, they can only include members in their structure, which are also static in their nature.

Static methods are uncommon and are not used often in a program. They have limited use, which is important considering that they are designed for a single-use situation. We once again stress that you do not need any object (instance) of a class to call one of its static methods. Let us look at the following example:

```
class DemoStatic {
    public static void copyVal(String st1, String st2) {
        st2=st1; //performs the copying of the first value to the second value
        System.out.println("The First value is: "+st1);
        System.out.println("The Second value is: "+st2);
    }
    public static void main(String[] args) {
        copyVal("ABC", "XYZ");
    }
}
```

We can clearly observe in this example that we have not created any object and have directly employed two static methods. We have first declared a static method `copyVal()` that copies one string argument to its other string argument. We then run the static method of `main()`, which runs the static method with two arguments. Our code changes the second string to have the first-string value. We get a console message of two lines which will both show the values of ABC. These methods are allocated in memory only once, whenever the compiler initiates the class.

2. **Non-static method:** Most methods that we use in Java are non-static methods. It means that they are employed within an object of a class. They provide the normal functional elements, which carry out the designed logic and create programming solutions in our code. These are the regular methods that we use in our programs. They can have their own memory instances each time they are called in our programming elements. Given below is a simple example of using non-static methods in Java programs:

```
public class DemoNonstatic {
    public void dispfunc() {
        System.out.println("Display using non-static method");
    }
    public static void main(String[] args) {
        DemoNonstatic obj=new DemoNonstatic();
        obj.dispfunc(); //directly using the dispfunc() will cause an error here
    }
}
```

Now, this prints the line in the quotations on the console. However, it is employed only when the newly created object of the `DemoNonstatic` class is created and calls this particular method. A new memory instance will be created for the use of method in such instances. Always remember that non-static methods are always called in the manner of `Object.methodname()` format.

11.3 | String Options

2

String entries are extremely important in the syntax of Java. These are several string options that are available in the language. String in Java is always initiated as a special class that presents all string data values as its instances. Remember, strings represent constant values in Java; therefore, they are only employed in a specific manner. Each change in value in actuality produces a new instance of String.

The String holds important value in Java since it is capable of representing any of the primitive wrapper classes that represent data values. When employed with the String class, other data values simply get converted to their string versions. The string can still be understood as an array of characters which can then provide the functionality of offering a fixed String instance. There are other important functions as well. Here, we present StringBuilder for this case.

The StringBuilder class objects are similar to normal String, but the main difference is that they can be modified. The class internally behaves as an array of characters, where different methods can be employed to change the content and the length of the stored sequence of characters. The StringBuilder Constructor has the ability to create a set of 16 elements. See the example below to understand how this class works:

```
…
StringBuilder sbr = new StringBuilder();
sbr.append("Greetings"); //adds 9 characters to the array
…
```

This code creates a `new StringBuilder` object and places 9 characters on the first 9 places in the total set of 16 elements. They occupy positions from 0 to 8 in their character array, within the internal workings of the class. It is an excellent class when you want to modify your strings and run operations that may alter its length and perform mathematical functions with some logic. It uses two special methods of `append()` and `insert()`, which give this class its edge and allows it to accept data from any primitive type.

Another related class is the "StringBuffer" class in Java. It can also be modified but has the additional capability of allowing for the use of concurrency. Java 9 and the subsequent versions are already going to emphasize a lot on using parallelism in Java programs and applications, where this class certainly has an improved application.

Flash ? Quiz What will be the output of multiplication of two String variables, String `variable1` holds a value of 10 and String `variable2` holds a value of 20?

All methods of this class are already synchronized and work in an overloaded manner when accepting any data. StringBuffer usually employs the same two methods of `append()` and `insert()` where both methods work the same for a new object instance. This class is different from the StringBuilder because it can accept any capacity. The overflow of the available buffer for this class automatically increases the available limit of the characters that can be stored in an object.

However, this makes the results difficult to control. You can use the StringBuilder class objects for swift operations. However, if you are creating a concurrent program where you want to ensure that you implement synchronization, StringBuffer objects will certainly provide you better control over the use of code application in a parallel manner.

QUICK CHALLENGE Create a comparison chart for StringBuffer and StringBuilder.

11.4 | Arrays

3

Arrays are an important part of Java as they are required for implementing most types of programs. They are produced as dynamic objects and are termed as the instances of the Object class. Arrays can be termed as container objects, where they can hold a fixed number of values belonging to a primitive class. However, it is also possible to declare an array of arrays. We will be covering arrays in more detail in Chapter 17. Arrays are declared using two parts:

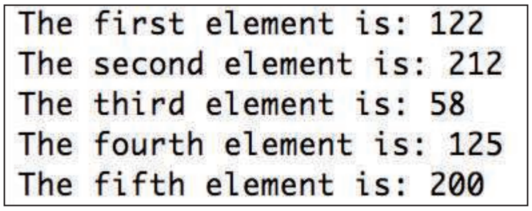
1. The first part describes the primitive type of data that the array will hold along with braced "[]" that indicate that this variable is an array.
2. The second element is the unique identifier, which is named according to the policies that we have described above.

Declaring an array simply creates a reference, where it cannot be used without having values. Using the "new" keyword as shown in the example below instantiates it according to the number of elements that we mention.

All values present in an array are termed as individual elements and are identified by a numerical index value. The first array location starts from "0" and then keeps moving forward according to the number of stored values. Let us look at how we use arrays in Java syntax:

```
package java11.fundamentals.chapter11;
public class DemoArray {
    public static void main(String[] args) {
        int[] oneArray;
        // we initiate an integer array
        oneArray = new int[5];
        //We set up 5 memory locations for the array
        oneArray[0] = 122;
        oneArray[1] = 212;
        oneArray[2] = 58;
        oneArray[3] = 125;
        oneArray[4] = 200;
        //We store 5 separate integers in successive positions
        System.out.println(“The first element is: ”+ oneArray[0]);
        System.out.println(“The second element is: ”+ oneArray[1]);
        System.out.println(“The third element is: ”+ oneArray[2]);
        System.out.println(“The fourth element is: ”+ oneArray[3]);
        System.out.println(“The fifth element is: ”+ oneArray[4]);
    }
}
```

The output from this program is simple. It will print out the following lines on the console.



```
The first element is: 122
The second element is: 212
The third element is: 58
The fourth element is: 125
The fifth element is: 200
```

As you can observe, it is an impractical approach to enter each value separately, especially when using large arrays. This is to just show an example of the syntax, as most times you will be running iteration loops that increment the array index positions to save values either directly or by using user input in interactive programs.

You can also create an array of all possible data types such as byte, short, long, float, double, boolean, char, and even Strings, which is kind of an array of arrays in itself. It is possible to place the brackets after the identifier, but the standard practice we will follow in this book is to always place them with the type of data. This means to always employ the practice of naming like `long[] largeNum` rather than `long largeNum[]`, although both are valid for the compiler.

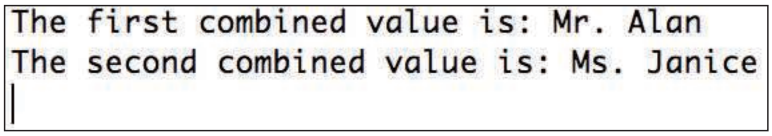
An easy method for placing values in a single step is also possible by using the following convention with the above example:

```
int[] oneArray = {122, 212, 58, 125, 200};
```

This will automatically set up an array with five elements, as shown by five values in the brackets, which must be separated by using commas. Java also allows the creation of multidimensional arrays. All string arrays are in fact multidimensional examples of character arrays. However, the difference in Java is that the components in such arrays are true arrays. This means that that array rows in Java can have different lengths. Given below is an example to show this property:

```
package java11.fundamentals.chapter11;
public class DemoArrayMulti {
    public static void main(String[] args) {
        String[][] salName = { { “Mr. ”, “Mrs. ”, “Ms. ” }, { “Alan”, “Janice” } };
        // We have elements for only two rows out of three
        System.out.println(“The first combined value is: ” + salName[0][0] + salName[1][0]);
        // Prints first name combination
        System.out.println(“The second combined value is: ” + salName[0][2] + salName[1][1]);
    }
}
```

The output for this program shows two lines.

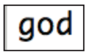


```
The first combined value is: Mr. Alan
The second combined value is: Ms. Janice
```

This example shows how we can use different number of elements in the arrays, as they act as fully functional arrays when employed in multiple dimensions. There are useful functions and methods that you can perform on arrays, such as writing `.length` to return the length of any given array. You can use the method of `arraycopy()`, which copies the contents of one array from the mentioned starting position to a destination array position. Given below is an example:

```
package java11.fundamentals.chapter11;
public class CopyArrayExam {
      static char[] copySource = { 'f', 'g', 'o', 'd', 'a', 'm', 'n' };
      static char[] copyDest = new char[3];
      public static void main(String args[]) {
         System.arraycopy(copySource, 1, copyDest, 0, 3);
         // Copy from first array location by reading from the second element
         // and placing in the second array from the first element, with a length
         // of three elements
         System.out.println(new String(copyDest));
   }
}
```

This program will print "god" on the console.



```
god
```

The copying process is simple to understand. The first argument in the method marks the source array, the second describes the starting position for copying array elements, the third describes the destination array, the fourth describes the starting position for pasting values, and the last position declares the length of the array to be copied.

There are other useful methods available in the java.util. Arrays class which are quite useful. They include methods such as `fill()`, `copyOfRange()`, `equals()` and various sorting methods. Java offers the powerful method of `parallelSort()`, which is faster by employing concurrency available of multiprocessor systems commonly available.

11.5 | Enums

Enums is another facility in Java that describe a special type of data which can define a collection of constant values in a program. There may be methods and other facilities which have been present in various Java versions. Enums allow programmers to be dynamic and set up any logic by creating a set of organized constants for use in a program.

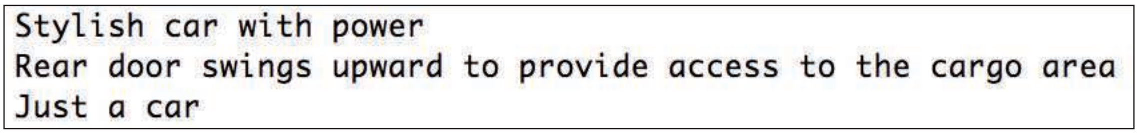
Take the example of directions, where there are four fixed values of north, east, south, and west. Similarly, the seven days of the week can form an enum as well, where these values are consistent and would never require a change during program executions. A typical instantiation in Java will occur in the following manner:

```
package java11.fundamentals.chapter11;
public enum CarTypes {Sport,Sedan,Hatchback,SUV,Mini,Hybrid}
```

Given below is an example that will help you understand how to use enum values in a Java syntax and how it is valuable to add functionality in your program by using the enum that was set up earlier:

```
package java11.fundamentals.chapter11;
public class EnumExample {
    CarTypes carTypes;
    public EnumExample(CarTypes carTypes) {
        this.carTypes = carTypes;
    }
    public void carFeatures() {
        switch (carTypes) {
            case Sport:
                System.out.println(“Stylish car with power”);
                break;
            case Hybrid:
                System.out.println(“Economical as partially runs on battery power”);
                break;
            case Hatchback:
            case SUV:
                System.out.println(“Rear door swings upward to provide access to the cargo
area”);
                break;
            default:
                System.out.println(“Just a car”);
                break;
    }
}
public static void main(String[] args) {
        EnumExample carOne = new EnumExample(CarTypes.Sport);
        carOne.carFeatures();
        EnumExample carTwo = new EnumExample(CarTypes.Hatchback);
        carTwo.carFeatures();
        EnumExample carThree = new EnumExample(CarTypes.Mini);
        carThree.carFeatures();
    }
}
```

This is a code which provides an excellent use of enums, while also describing situations where we can implement case and break scenarios that Java brings in from C and C++ languages. This program will print three lines with three created objects that all run the single defined method of `carFeatures()`. Given below is the output.



```
Stylish car with power
Rear door swings upward to provide access to the cargo area
Just a car
```

All created enums are the extensions of *java.lang.Enum*, which means that there cannot be a further inheritance of state. Creating an enum will not extend anything other than the single instance of the set of fixed values. There are plenty of ways in which you can use these enums. Remember to always treat them as a constant set of values, and you will be able to employ them in a variety of application settings when you are using the Java syntax for program developments.

Enums offer excellent advantages, and therefore you should employ them in your program syntax wherever applicable. Listed below are some important benefits of using enums for extending a set of constant values:

1. Enums offer excellent safety when you want to ensure that the correct type of data will be available in a particular variable.
2. It is possible to traverse your Java enums providing excellent functionality.
3. As shown in the example, enum objects are perfect for use in “switch” case representations where you want to implement specific responses in a program, according to a specific but fixed set of situations.
4. It is possible to have various constructors, methods, and fields that represent an enum.
5. Enums are perfect for implementing interfaces. However, they do not perform extensions and therefore, offer programming safety.

Enums provide an excellent set of internal methods that are always available once an enum has been set up in a program. They provide excellent additional functionality such as the ability to return the enum values.

QUICK CHALLENGE Write real life use cases where you will need enums.

11.6 | Wrapper Classes

5

Java provides various wrapper classes that are useful in the following ways:

1. **Wrap primitive values in an object:** As you know, primitives are not objects and hence they cannot be directly included in the activities that are reserved for objects, activities like adding itself to Collections. Collections can only take objects and hence primitives cannot get added.
2. **Utility functions for primitives:** Various functions are not available to primitives such as converting to and from String objects, converting to and from binary, octal and hexadecimal.

11.6.1 Outline of Wrapper Classes

Wrapper class, as name suggests, wraps a primitive in an object. Each primitive has a corresponding wrapper class. For example, int primitive has Integer wrapper class, float primitive has Float wrapper class, etc. Table 11.2 shows primitives and their equivalent wrapper classes.

Table 11.2 Primitives and their equivalent wrapper classes

Primitive	Wrapper Class	Constructor Argument
byte	Byte	byte or String
int	Integer	int or String
short	Short	short or String
long	Long	long or String
float	Float	float, double, or String
double	Double	double or String
char	Character	char
boolean	Boolean	boolean or String

11.6.2 Creation of Wrapper Objects

Wrapper objects can be created in three different ways: Constructor, valueOf(String val), and valueOf(String val, int radix).

1. **Constructor:** There are two types of Constructors other than Character Wrapper. One takes a primitive value and other takes a String representation of primitive. See the following examples to understand it better.
   Primitive examples are as follows:
   - `Integer i = new Integer(10);`
   - `Float f = new Float(5.5f);`
   - `Boolean b = new Boolean(true);`

   String representation examples are as follows:
   - `Integer i = new Integer("10");`
   - `Float f = new Float("5.5f");`
   - `Character c = new Character('c');`
   - `Boolean b = new Boolean("true");`

   Please note that in String representation, Character only provides one constructor which accepts char as an argument.

2. **valueOf(String val):** This type accepts one argument as String. For example:

```
Float f = Float.valueOf("5.5f");
```

3. **valueOf(String val, int radix):** This type accepts two arguments. One is String and other one is radix. For example;

```
Integer i = Integer.valueOf("1010",2);
```

11.6.3 Wrapper Conversion Utility Methods

One of the ways in which Wrapper is useful is the utility methods to convert primitives. There is a common way of converting to a primitive. We can use the `xxxValue()` format, where `xxx` can be changed to the desired primitive type. Table 11.3 shows some common conversion methods to convert String to a primitive.

Table 11.3 Common conversion methods to convert String to a primitive

	Boolean	Byte	Character	Double	Float	Integer	Long	Short
byteValue		x		x	x	x	x	x
doubleValue		x		x	x	x	x	x
floatValue		x		x	x	x	x	x
intValue		x		x	x	x	x	x
longValue		x		x	x	x	x	x
shortValue		x		x	x	x	x	x

Let us try to understand this with the help of following examples:

1. Create a wrapper object:

```
Integer i = new Integer(42);
```

2. Convert Integer *i* value to byte:

```
byte b = i.byteValue();
```

3. Covert Integer *i* value to short:

```
short s = i.shortValue();
```

4. Covert Integer *i* value to double:

```
double d = i.doubleValue();
```

Table 11.4 shows common conversion methods to convert Wrapper to a primitive.

Table 11.4 Common conversion methods to convert Wrapper to a primitive

		Boolean	Byte	Character	Double	Float	Integer	Long	Short
parseXxx	Static Method Throws NumberFormatException		x		x	x	x	x	x
parseXxx (with radix)	Static Method Throws NumberFormatException		x				x	x	x

Let us try to understand with the help of the following examples:

1. `double d = Double.parseDouble("5.5");`
2. `long l = Long.parseLong("101010", 2);`

Table 11.5 shows common conversion methods to convert String to a Wrapper.

Table 11.5 Common conversion methods to convert String to a Wrapper

		Boolean	Byte	Character	Double	Float	Integer	Long	Short
valueOf	Static Method Throws NumberFormatException		x	x		x	x	x	x
valueOf (with radix)	Static Method Throws NumberFormatException			x				x	x

Let us try to understand with the help of the following examples:

1. `Double d = Double.valueOf("5.5");`
2. `Long l = Long.valueOf("101010", 2);`

Table 11.6 shows common conversion methods to give String representation.

Table 11.6 Common conversion methods to give String representation

		Boolean	Byte	Character	Double	Float	Integer	Long	Short
toString		x	x	x	x	x	x	x	x
toString (primitive)	Static Method	x	x	x	x	x	x	x	x
toString (primitive, radix)	Static Method						x	x	

Let us try to understand with the help of the following example:

```
Double d = new Double ("5.5");
System.out.println("d = "+ d.toString() );
```

The above code prints the following result:

```
d = 5.5
```

11.7 | Autoboxing and Unboxing

We have already discussed that the Java compiler is capable of moving elements between different types of primitive datasets. After Java 5, Java compiler is capable of automatically converting the primitive types into their object Wrapper classes. As shown in Figure 11.1, a process of compiler converting from primitive datatype to Wrapper class called *autoboxing* and from Wrapper class to primitive datatype is called *unboxing*.

In other words, we can say that, the compiler is capable of creating these Wrapper classes and this ability is termed as autoboxing. If this option was not present in the operating language, there would be an error every time a method required an object that should represent a Wrapper class in Java. For example,

```
Character ch = 'X';
```

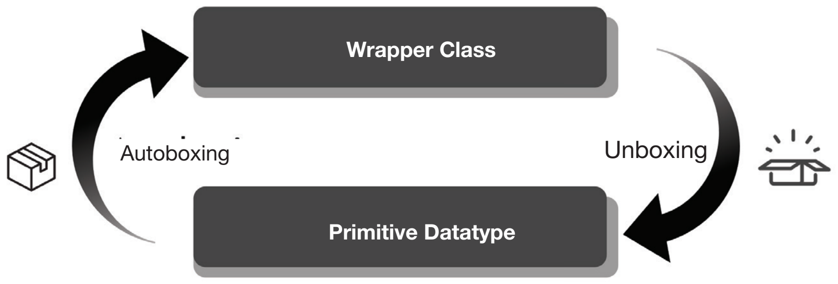


Figure 11.1 Autoboxing and unboxing process diagram.

This explains that there is a primitive data of character type which gets changed into a Character object using the generics syntax. Here is another way in which this functionality is achieved:

```
package java11.fundamentals.chapter11;
    import java.util.ArrayList;
    import java.util.List;
public class AutoboxingExample {
    public static void main(String args[]){
        List<Integer> alist = new ArrayList<>();
        for (int i = 1; i < 10; i =+ 1){
            alist.add(i);
        }
    }
}
```

This code reads the primitive integer values and then converts them to Integer objects to ensure that the List can be maintained which is a collection of objects. There are two ways in which primitive to Wrapper auto conversion takes place. The first way is when a value is passed as a parameter where the method definition requires a Wrapper class object. The second way is when a value is assigned to a variable of a specific Wrapper class.

The Java compiler is capable of doing the opposite of this function as well. It can take the example of a Wrapper class object and convert it into primitive data types because they are required to run the required arithmetic operators. This process of creating data from an object is the reverse of the earlier process and is termed as unboxing. Let us take a look at an unboxing example,

```
package java11.fundamentals.chapter11;
import java.util.ArrayList;
import java.util.List;
public class UnboxingExample {
    public static void main(String args[]){
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(1);
        nums.add(15);
        nums.add(20);
        System.out.println("Total is " + addNumbers(nums));
    }
    public static int addNumbers(List<Integer> nums) {
        int total = 0;
        for (Integer num: nums){
            total += num;
        }
        return total;
    }
}
```

Here, we can see in the code that we take values that are, in fact, integer objects and then perform addition operation on them. We know that they cannot be run on objects, but only on integer type primitive data values. The compiler does not return an error here and simply converts the objects to the required integer type primitive data. This code produces the following output:

```
Total is 36
```

This happens because the code is internally changed by the compiler. What happens is that the value stored in the object is read by the compiler using the `intValue()` method and is then used during the program. This is an excellent example of unboxing, because it finds the primitive value which is contained within the box of a Wrapper class object.

The language is therefore, often termed as a semi object-oriented language, because it is still capable of converting primitive data types and is not limited by only using objects for all functions. The relevant conversions occur in a smart manner, always providing the required functionality.

11.8 | Developing Logic

Developing logic is possible when we first understand the different programming tools that we can employ in Java. These tools include arithmetic operations, comparators, and the "if" based several types of statements. There are also loops which are great for developing simple logic and reducing the number of lines of code required to perform a specific task.

We will be discussing these elements and how their syntax is carried out in Java. We will start with the basic operators then move towards comparisons and loops, including nested looping programs.

11.8.1 Various Types of Operators

7

In the earlier Section 11.1.2, we have seen variables like int, float, double, char, String, etc. Variables are placeholders of specific types of data. For example, int holds numbers and String holds characters. For various numeric values such as int, float, and double, you are going to need to add them together, increment them, compare one to another, multiply them, divide them, etc. Java provides operators for these types of operations. An operator takes two values to generate result. These values are called *operands*. The operator stays in the middle of two operands. For example, the "+" operator adds two numeric values 4 and 6 to produce 10. So, the placement would look like "4 + 6". "4" is the left operand, followed by "+" operator and then "6" operand. Figure 11.2 shows various operators supported by Java.

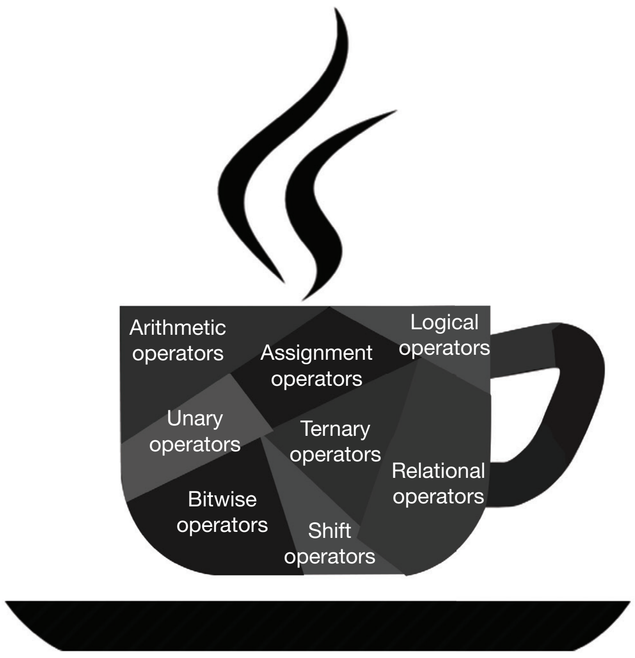


Figure 11.2 Operators in Java.

In C++, the operators are overloaded but this is not the case in Java. However, there are a few exceptions to this, where a few operators come overloaded in Java. For example, the "+" operator can be used to add two numeric primitives, or concatenate two operands if they are Strings. Similarly, the |, ^, and & operators can be used in two different ways.

11.8.1.1 Assignment Operators

As the name suggest, assignment operator is used to assign a value to a variable. It may look like `int a = 10`. It may be simple but there will be situations which will demand complex expressions and casting. These involve primitive and reference variable assignments. Now let us understand what is variable. Variable is nothing but a bit holder with a specific type. For example, int holder, double holder, Boolean holder, and String holder. Each holder contains bits that represent value. Like for primitives, the bits represent numeric value. Similarly, objects also hold certain value. We will learn about object assignments in Section 11.7.1.2. Now, let us learn about primitive assignments.

11.8.1.1.1 Primitive Assignments

Similar to Math, Java uses "=" as an assignment operator to assign a value to a variable. There are various other types of assignment operators used in Java. A primitive variable can be assigned by a literal or an expression. For example,

```
1. int a = 10; //literal assignment
2. int b = a * 10; // assignment with an expression including a literal
3. int c = a + b; // assignment with an expression
```

In the first case of `int a = 10`, a literal integer 10 is implicitly an int. Now, let us explore some tricky cases. We all know that int is a 32-bit value and long is 64-bit. Hence, just as a bigger bucket can hold a smaller bucket, similarly long can hold an int value. Now, we all know that byte is an 8-bit value which is definitely smaller than int. However, what will happen if you try to change the type of a variable? For example,

```
byte a = 10;
```

Will this give us a compiler error? No. This will not give a compiler error as the literal value will be automatically cast by the compiler. The compiler will see the above code as follows,

```
byte a = (byte) 10; // Explicitly cast the int literal to a byte
```

This is also true for char and short type.

11.8.1.1.2 Primitive Casting

As we have seen in the above example, casting converts primitive values from one type to another. Casting can be implicit or explicit as we have seen earlier. Implicit cast means the compiler takes care of the conversion where you do not have to write code for the cast. Whereas in the explicit casting, you have to specifically write code for the cast. Implicit casting occurs only when you try to put smaller values into bigger ones like int into long, byte into int, etc. If you do not mention the casting then in the other way conversion compiler will through "possible loss of precision" error. This conversion like big value into smaller container for example long into int, int into byte etc. This type of conversion is called *narrowing*, which requires explicit cast. Let us look at some implicit and explicit casts.

Examples of implicit cast:

```
1. int a = 10;
2. long b = a; // Implicit cast, an int value always fits in a long
```

Examples of explicit cast:

```
1. float f = 134242.003f;
2. int c = (int) f; // Explicit cast. In this case, we will lose the precision
```

The following example is going to give a compiler error as it is expecting an explicit casting,

```
package java11.fundamentals.chapter11;
public class CastingError {
   public static void main(String args[]){
      int a = 10.000; //going to through error
   }
}
```

The above program gives the following result.

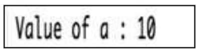
```
Exception in thread “main” java.long.Error: Unresolved compilation problem:
         Type mismatch: cannot convert from double to int


at java11.fundamentals.chapter1.CastingError.main(CastingError.java:5)
```

To solve the above problem, you can cast the floating-point value explicitly to int but you are going to lose the precision.

```
package java11.fundamentals.chapter11;
public class CastingErrorSolution {
   public static void main(String args[]){
      int a = (int) 10.000; //this will work
      System.out.println(“Value of a : ” + a);
   }
}
```

The above program produces the following result.

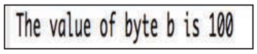


```
Value of a : 10
```

As you can see, the above casting results in loss of digits after the decimal. Now, let us see a few interesting cases. What if you cast a larger type to the smaller one? For example, casting long to byte. Let us study this with the following program and its output.

```
package java11.fundamentals.chapter11;
public class LargeToSmallCast {
   public static void main(String [] args) {
      long l = 100L;
      byte b = (byte)l;
      System.out.println(“The value of byte b is ” + b);
   }
}
```

The above code produces the following result.

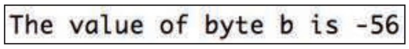


```
The value of byte b is 100
```

Now let us increase the value of long variable, which is larger than the container size of byte. Let us see what will happen if we set value of long as 200.

```
package java11.fundamentals.chapter11;
public class LargeToSmallCastOuterRange {
    public static void main(String [] args) {
        long l = 200L; //this will give us unexpected result
        byte b = (byte)l;
        System.out.println("The value of byte b is " + b);
    }
}
```

The result shows that the code is giving us an unexpected result.



```
The value of byte b is -56
```

Why do we get a negative value? This happened because the value of long, which is narrowed, is too large for byte to hold. In this case, bits to the left of lower 8 go away. And after the removal of the lower 8 bits, if the value of the leftmost bit, which is the sign bit, turns out to be 1 then the primitive gives a negative value.

11.8.1.2 Object Assignments

We now understand if we say `int a = 10`, it means "*a* as int type holds value of 10 but in bits form". But in the object world it is a little different. It is not the direct assignment of the value but a reference. For example, object *b* in the following example does not contain Book object itself but a reference to the object that sits on the heap. Hence, it is called *reference variable*.

```
Book b = new Book();
```

The above code does the following three things:

1. It creates a reference variable *b* of type Book.
2. It then creates a new Book object on the heap.
3. It assigns the newly created Book object to the reference variable *b*.

You may also assign a null value to the object by specifying it in the following way.

```
Book b = null;
```

In this case, the variable *b* holds bits representing "null". This means that it creates space for the Book reference variable but does not actually create a Book object. This can be done by not explicitly referring to "null" as well. For example, `Book b;`

Please note that in method scope you must assign a value to a variable whereas in class scope if value is not assigned then variables get their default value. For example, `Book b;` can only be specified on the class level and not in any method. Variables inside methods called as local variables and do not carry any default values.

In the object-oriented world, we have a concept of superclass and subclass. Although you will be learning a lot about OOP in the Chapter 12, for this object reference concept, it is important for you to know a little bit about superclass and subclass. In Java, every class must originate from some class, which is called *superclass*. For example, a class called "Animal" can act like a superclass and holds standard animal properties, from which all other animals will be derived. Classes "Dog" and "Cat" can be derived from the superclass "Animal" so they will inherit all the properties of class "Animal". Now that you understand the concept of superclass and subclass, we should try to understand the pool of object references. In this world, a superclass object can refer to any object of its subclasses. For example, the Animal object reference can be used to refer to Dog object.

```
package java11.fundamentals.chapter11;
public class Animal {
   public void makeSound() {
      System.out.println("Animal makeSound method");
   }
}
```

The above code declares an Animal class. This class contains a method makeSound().

```
package java11.fundamentals.chapter11;
public class Dog extends Animal{
   public void makeSound() {
         System.out.println("Dog makeSound method");
   }
}
```

The above code declares a Dog class which extends Animal class. In this case, Dog is a subclass of the Animal class. In other words, Animal is the superclass of Dog.

```
package java11.fundamentals.chapter11;
public class ObjectReferenceExample {
   public static void main (String [] args) {
      Animal dogObj = new Dog(); // Legal declaration as Dog is a subclass of Animal
      Dog animalObj = new Animal(); // Illegal declaration as Animal is not a subclass
of Dog
   }
}
```

The above program gives the following result.

```
Exception in thread "main" java.long.Error: Unresolved compilation problem:
        Type mismatch: cannot convert from Animal to Dog


at java11.fundamentals.chapter1.ObjectReferenceExample.main(ObjectReferenceExample.java:6)
```

As you can see in the output, the error says, "Type mismatch: cannot convert from Animal to Dog". It means you cannot assign a superclass object to a subclass variable. You can only assign a subclass object to a superclass variable. This is because a subclass Dog object is guaranteed to be able to do anything that a superclass Animal object can do. In other words, due to this superclass reference used for a Dog object, any variable with within a Dog object can invoke Animal methods as this object holds the Animal reference. For example, a Dog object can invoke the `makeSound()` from the Animal class. However, in this case, we have overridden `makeSound()` method in the Dog class so with a Dog object, even with the Animal reference, the inherited method from Dog class will get called. See the following example,

```
package java11.fundamentals.chapter11;
public class ObjectReferenceMethodCallExample {
    public static void main (String [] args) {
        Animal dogObj = new Dog(); // Legal declaration as Dog is a subclass of Animal
        dogObj.makeSound();
    }
}
```

The above program gives the following result which shows that the Dog object's `makeSound()` method has been called.

```
Dog makeSound method
```

If we don't override `makeSound()` method in the Dog class, upon invoking `makeSound()` on the Dog object will invoke the Animal class's `makeSound()` method. See the following example:

```
package java11.fundamentals.chapter11;
public class DogWithoutOverriden extends Animal{
}
```

In the above program, we have not added `makeSound()` implementation and hence, it will always use the Animal class `makeSound()` implementation.

```
package java11.fundamentals.chapter11;
public class ObjectReferenceMethodCallExample2 {
    public static void main (String [] args) {
        Animal dogObj = new DogWithoutOverriden(); // Legal declaration as Dog is a
subclass of Animal
        dogObj.makeSound();
    }
}
```

The above program produced the following output which shows that the Animal class's `makeSound()` method has been invoked.

```
Animal makeSound method
```

However, think for a second, if the actual object is Animal and reference object is Dog, it would not know which `makeSound()` method to call. Hence this reverse assignment like subclass reference and superclass object does not work.

11.8.1.3 Arithmetic Operators

Arithmetic operators are mainly used to perform mathematical related operations. They are useful in performing operations like addition, subtraction, multiplication, division, and remainder. These operations are permitted on numeric data types like byte, short, int, long, float, and double. Table 11.7 shows the Arithmetic operators and their meanings.

Table 11.7 Arithmetic operators and their meanings

Operator	Meaning
+	**Addition operator** – Use for adding two numbers
–	**Subtraction operator** – Use for subtracting two numbers
*	**Multiplication operator** – Use for multiplying two numbers
/	**Division operator** – Use for dividing two numbers
%	**Remainder operator** – Use for getting remainder of two numbers

The following example demonstrates the use of these operators.

```
package java11.fundamentals.chapter11;

public class ArithmeticOperators {
    public static void main(String args[]){
        int x = 20;
        int y = 15;

        ArithmeticOperators ao = new ArithmeticOperators();
        int additionResult = ao.doAddition(x,y);
        int subtractionResult = ao.doSubtraction(x, y);
        int divisionResult = ao.doDivision(x, y);
        int multiplicationResult = ao.doMultiplication(x, y);
        int remainderResult = ao.doRemainder(x, y);
        System.out.println("Addition Result = " + additionResult + "\nSubtraction Result
= " + subtractionResult + "\nDivision Result = " + divisionResult + "\nMultiplication
Result = " + multiplicationResult + "\nRemainder Result = " + remainderResult);
    }
    public Integer doAddition(int x, int y){
        return x + y;
    }
    public Integer doSubtraction(int x, int y){
        return x - y;
    }
    public Integer doDivision(int x, int y){
        return x / y;
    }
    public Integer doMultiplication(int x, int y){
        return x * y;
    }
    public Integer doRemainder(int x, int y){
        return x % y;
    }
}
```

11.8.1.4 Unary Operators

Unary operators perform various operations such as increment, decrement, inverting the value of a Boolean, and negating an expression. It is quite convenient and widely used in loops. Table 11.8 shows the unary operators Java provides.

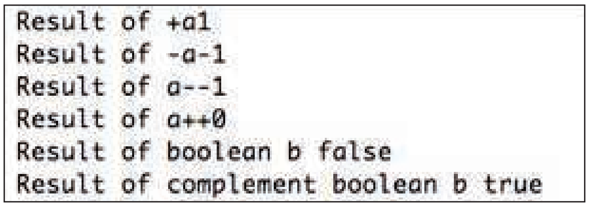
Table 11.8 Unary operators and their meanings

Operator	Meaning
+	**Unary plus –** It shows number as positive, but it is not needed as numbers are already positive
–	**Unary minus –** It inverts the sign of an expression
++	**Increment operator –** It increments value by 1
--	**Decrement operator –** It decrements value by 1
!	**Logical complement operator –** It inverts the value of a Boolean

The following program shows the use of the Unary operators.

```
package java11.fundamentals.chapter11;
public class UnaryOperators {
   public static void main(String args[]){
      int a = 1;
      boolean b = false;
      System.out.println("Result of +a" + +a);
      // result is now -1
      System.out.println("Result of -a" + -a);
      // result is now 0
      System.out.println("Result of a--" + a--);
      // result is now 1
      System.out.println("Result of a++" + a++);
      // false
      System.out.println("Result of boolean b " + b);
      // true
      System.out.println("Result of complement boolean b " + !b);
   }
}
```

The above program produces the following result.



```
Result of +a1
Result of -a-1
Result of a--1
Result of a++0
Result of boolean b false
Result of complement boolean b true
```

11.8.1.5 Equality and Relational Operators

In many situations, as a programmer you need to check the equality of two operands to see if one is greater than, less than, equal to, or not equal to another operand. For this use case Java provides various operators, which are called equality and relational operators. These operators give result in Boolean, which is either true or false, and they are mainly used in decision-making and loops. Table 11.9 shows the equality and relational operators.

Table 11.9 Equality and relational operators

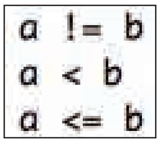
Operator	Description	Example
==	equal to	10 == 20 is evaluated to false
!=	not equal to	10 != 20 is evaluated to true
>	greater than	10 > 20 is evaluated to false
<	less than	10 < 20 is evaluated to true
>=	greater than or equal to	10 >= 20 is evaluated to false
<=	less than or equal to	10 <= 20 is evaluated to true

The example below shows the use of these operators.

```
package java11.fundamentals.chapter11;

public class EqualityRelationalOperators {
   public static void main(String args[]) {
      int a = 10;
      int b = 20;
      if (a == b) {
         System.out.println("a == b");
      }
      if (a != b) {
         System.out.println("a != b");
      }
      if (a > b) {
         System.out.println("a > b");
      }
      if (a < b) {
         System.out.println("a < b");
      }
      if (a <= b) {
         System.out.println("a <= b");
      }
   }
}
```

The above program produces the following result.



```
a != b
a < b
a <= b
```

11.8.1.6 The instanceof Operator

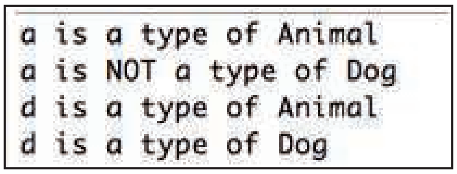
The instanceof operator is useful in situations where you want to check if an object was instance of the specified class. As we have seen earlier, one of the OOP properties is inheritance, in which the subclass can be referred by a superclass reference. In such a scenario, it is sometimes important to check if the object belongs to the class you are expecting. In this case, the instanceof operator becomes useful, and it tells you if the comparing object is type of the class you are expecting.

The following example demonstrates the use of the instanceof operator. We will use the example classes "Animal" and "Dog" that we used earlier to demonstrate superclass–subclass relationship.

```
package java11.fundamentals.chapter11;

public class InstanceofOperator {
    public static void main(String args[]){
        Animal a = new Animal();
        Dog d = new Dog();
        if(a instanceof Animal){
            System.out.println("a is a type of Animal");
        }else{
            System.out.println("a is NOT a type of Animal");
        }
        if(a instanceof Dog){
            System.out.println("a is a type of Dog");
        }else{
            System.out.println("a is NOT a type of Dog");
        }
        if(d instanceof Animal){
            System.out.println("d is a type of Animal");
        }else{
            System.out.println("d is NOT a type of Animal");
        }
        if(d instanceof Dog){
            System.out.println("d is a type of Dog");
        }else{
            System.out.println("d is NOT a type of Dog");
        }
    }
}
```

The above program gives the following result.



```
a is a type of Animal
a is NOT a type of Dog
d is a type of Animal
d is a type of Dog
```

11.8.1.7 Logical Operators

Often times we come to a point in our program where we need to match various conditions to make a decision. For example, you may want to return a value if it falls between 10 and 20. In this case, you need an operator that will compare your value with two other values. This can be done with the help of logical operators. Table 11.10 shows all the logical operators and their descriptions. Logical operators return Boolean result, which helps compute the comparison. There are six logical operators: *&* (AND), | (OR), ^ (Xor), *!* (Not), || (conditional-OR), and *&&* (conditional-AND). In the example below, we compare the value, say *x*, with 10 and 20.

```
if(x >= 10 && x <= 20){
return x;
}
```

These types of expressions called *compound expressions* as they let us combine two or more conditions into a single expression.

Table 11.10 Logical operators

Operator	Description
||	**conditional-OR** – It returns true if either of the Boolean expression is true
&&	**conditional-AND** – It returns true if all Boolean expressions are true
!	**Not** – It returns false if true is passed and true if false is passed
|	**OR** – It works similar to conditional-OR, where it returns true if at least one of the operands evaluates to true. But the difference is it evaluates both the operands before checking the OR condition.
&	**AND** – It works similar to conditional-AND, where it returns true if both of the operands evaluate to true. But the difference is it evaluates both the operands before checking the AND condition
^	**Xor** – It returns true if only one of the operands evaluates to true

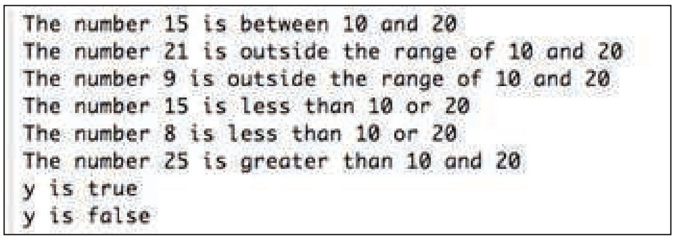
Notes:

1. False || true expression is evaluated to true.
2. False && true expression is evaluated to false.

The following example shows the uses of logical operators.

```
package java11.fundamentals.chapter11;
public class LogicalOperators {
    private int a = 10;
    private int b = 20;
    public static void main(String args[]){
        LogicalOperators lo = new LogicalOperators();
        lo.checkBetweenAB(15);
        lo.checkBetweenAB(21);
        lo.checkBetweenAB(9);
        lo.checkLessAOrLessB(15);
        lo.checkLessAOrLessB(8);
        lo.checkLessAOrLessB(25);
        lo.checkLogicalNot(true);
        lo.checkLogicalNot(false);
    }
    public void checkBetweenAB(int x){
        if(x >= a && x<=b){
            System.out.println("The number " + x + " is between " + a + " and " + b);
        }else{
            System.out.println("The number " + x + " is outside the range of " + a + " and "
+ b);
        }
    }
    public void checkLessAOrLessB(int x){
        if(x < a || x < b){
            System.out.println("The number " + x + " is less than " + a + " or " + b);
        }else{
            System.out.println("The number " + x + " is greater than " + a + " and " + b);
        }
    }
    public void checkLogicalNot(Boolean y){
        if(y){
            System.out.println("y is true");
        }
        if(!y){
            System.out.println("y is false");
        }
    }
}
```

The above program produces the following result.



```
The number 15 is between 10 and 20
The number 21 is outside the range of 10 and 20
The number 9 is outside the range of 10 and 20
The number 15 is less than 10 or 20
The number 8 is less than 10 or 20
The number 25 is greater than 10 and 20
y is true
y is false
```

As you can see, the logical AND operator makes sure that both the conditions are true whereas the logical OR operator checks if either of the conditions is true. In the OR check scenario, if the first condition satisfies then it will not check for the second condition and simply return the result. In the AND check scenario, it will check for the second condition only if the first condition satisfies. If the first condition does not satisfy then it will break the "if" block.

11.8.1.8 Ternary Operator

The ternary operator allows you to shorten the if-then-else statement. Although the code is less readable than the actual if-then-else block, it helps to write code faster. The syntax is,

```
variable = Expression ? value1 : value2;
```

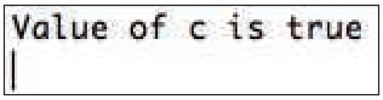
This is a handy operator and it is very easy to understand. If the Expression is true, `value1` is assigned to `variable` and if the Expression is false, `value2` is assigned to `variable`. Let us see an example,

```
package java11.fundamentals.chapter11;
public class TernaryOperator {
    public static void main(String args[]){
        int a = 10;
        int b = 20;
        boolean c;

        //Following expression checks if the value of a is less than b
        c = a < b ? true : false;

        System.out.println("Value of c is " + c );
    }
}
```

The above program produces the following result.



```
Value of c is true
```

11.8.1.9 Bitwise and Bit Shift Operators

Bitwise and bit shift operators are a bit uncommon for use in programs. However, they are very useful in some situations. As the name suggests, these operators perform bitwise and bit shift operations on integral types. Table 11.11 lists the bitwise and bit shift operators.

Table 11.11 Bitwise and Bit shift operators

Operator	Description
~	**Bitwise complement –** It inverts the bit pattern, such as making every "0" to "1" and every "1" to "0"
<<	**Left shift –** It shifts a bit pattern to left
>>	**Right shift –** It shifts a bit pattern to right
>>>	**Unsigned right shift –** It shifts a zero into the leftmost position
&	**Bitwise AND –** It performs bitwise AND operation
^	**Bitwise exclusive OR –** It performs bitwise exclusive OR operation
|	**Bitwise inclusive OR –** It performs bitwise inclusive OR operation

The following example demonstrates the use of bitwise and bit shift operators.

```
package java11.fundamentals.chapter11;

public class BitwiseBitshiftOperators {

public static void main(String[] args) {

       int x = 30;
       int y = 60;
       int z = 300;
       int k = 55;
       int l = 15;
       int output;

       output = ~k; //performs bitwise complement operation. It simply inverts the bit
pattern by changing every 0 to 1, and every 1 to 0.
       System.out.println(output);

       output = x ^ y; //performs bitwise Xor operation where it compares the close by
bits. If found same, it returns 0 or else 1.
       System.out.println(output);

       output = x & y; //performs bitwise AND operation where it compares the close by
bits. If found both bits as 1, it returns 1. If either of the bit is not 1 then 0.
       System.out.println(output);

       output = x | y; //performs bitwise OR operation where it compares the close by
bits. If found either of the bit 1, it returns 1 or else 0.
       System.out.println(output);

       //perform signed left shift operation. It shifts a bit pattern to the left by
number of specified bits. The low-order positions bits are filled with zero bits.
       System.out.println(z << 0);
       System.out.println(z << 2);
       System.out.println(z << 6);

       //perform signed right shift operation. It shifts a bit pattern to the right by
number of specified bits.
       System.out.println(z >> 0);
       System.out.println(z >> 1);
       System.out.println(z >> 6);

       System.out.println(l >>> 1); //perform unsigned right shift operation
   }
}
```

QUICK CHALLENGE Create a chart to show all the operators with examples.

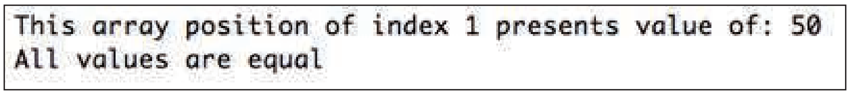
11.8.2 Expressions

Expressions are a collection of methods, variables, and operators that are constructed according to Java's specific syntax to evaluate values and provide answers. In all the examples in this chapter, we have often used these expressions, which we are defining here in terms of their role in creating logical functionality in a Java program. Here, we present a few expressions that use different operators and methods to cover all types of instances.

```
1. package java11.fundamentals.chapter11;
2. public class ExpressionsExample {
3.     public static void main(String args[]){
4.         int num = 0;
5.         int num1 = 1;
6.         int num2 = 2;
7.         int val1 = 4;
8.         int val2 = 4;
9.         int arrayOne[] = new int[10];
10.        arrayOne[1] = 50;
11.        System.out.println("This array position of index 1 presents value of: " +
           arrayOne[1]);
12.        int res = num1 + num2;
13.        if (val2==val1){
14.        System.out.println("All values are equal");
15.        }
16.    }
17. }
```

These are a few instances which describe how to set up expressions. Setting expressions are important because they often control the statements that come after them, depending on the logic which was generated and processed by the expression. Line 4 code shows the correct initialization of a variable num which is then equated to an integer of 0. On line 10, a value is saved in the second indexed location of an array, which is then printed on the console in the next line of code.

Line 12 presents a mathematical operation where two numbers are added and their sum is stored in another variable of type integer. Since the result is of integer type, we may find that in an actual program, there is a need for type promotion if `num1` and `num2` are not of matching datatypes. On line 13, we perform a logical comparison of two variables and then only on line 14 print a message to the console, if both variables have equal values. Following image shows the result of the above program.



```
This array position of index 1 presents value of: 50
All values are equal
```

Expressions are important as they provide a controlling logic. The order of operations can often be ambiguous if we do not follow ideal mathematical practices. Therefore, it is important to understand the best ways to present mathematical operations, just as we would present them traditionally. Take the example of using brackets to describe clear information, as given in the examples below.

```
(x+y)/n;
```

In the above example, as per the operator precedence, Java will first compute the addition and then division by `n`.

```
(x/n)+y;
```

In the above example, as per the operator precedence, Java will first compute the division and then addition with `y`.

```
X+(y/n);
```

In the above example, as per the operator precedence, Java will first compute the division of `y` by `n` and then addition with `x`.

The use of simple brackets in the above examples provide an ideal control over the results of the mathematical operations. There can be a number of brackets and order of operations that always ensure that it is not possible to have a wrong result, which can destroy the program logic altogether.

11.9 | Control Flow

8

Control flow is an important element in any programming language. The Java syntax also allows programmers to use statements which ensure that the program direction and execution can be controlled based on the choices available during program execution. Figure 11.3 shows all the control statements. We will discuss all these different ways in which the control statements can be used. Below we will see how "if" can provide control, similarly how "switch" can be used to control the flow which we have already seen in Section 11.5.

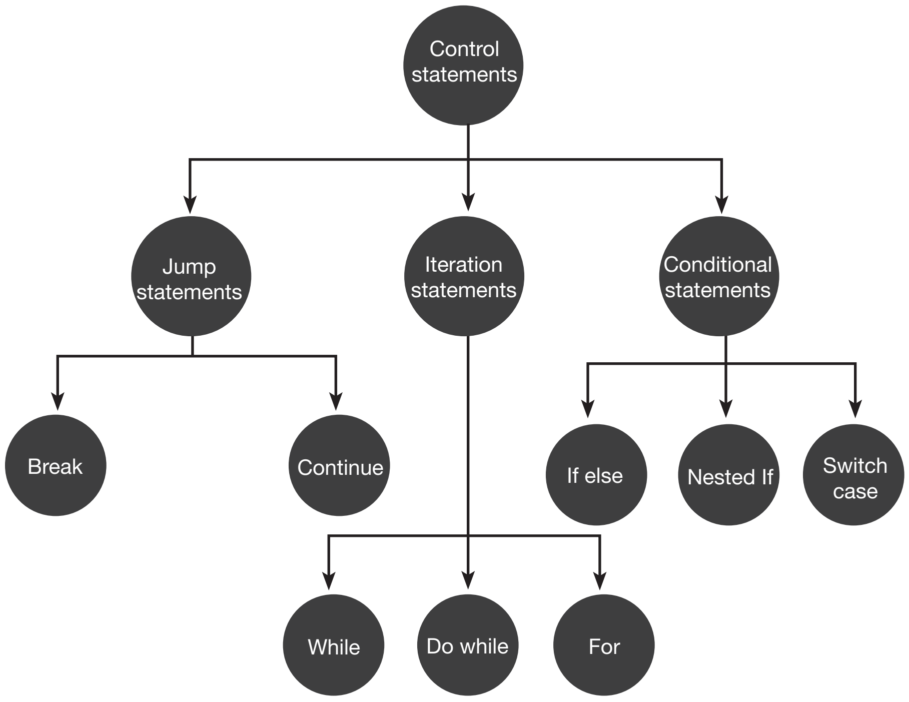
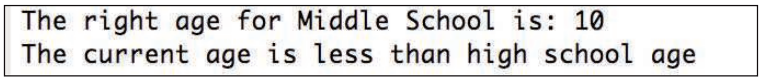


Figure 11.3 Control statements.

The "if" statement can be termed as a legacy statement, as it is present universally in almost all programming languages. It provides the basic control flow by ensuring that a particular program function is only performed if the tested condition, which is predefined in the statement, is found to be true by the compiler. Below is a simple example which shows its individual use.

```
package java11.fundamentals.chapter11;
public class AgeCompare {
    public static void main(String[] args) {
        int age = 10;
        System.out.println("The right age for Middle School is: " + age);
        if (age < 14) {
            System.out.println("The current age is less than high school age");
        }
    }
}
```

The above program is excellent in providing the control flow use. The system sets up a variable fixed at the value of 10. When the if statement runs and finds that the age is less than 14, it executes the next code block. It is also possible to represent a single statement without using the brackets. However, the ideal Java syntax practice is to use the brackets to create code blocks. In most situations, you will need to carry out a set of functions if the condition is found to be true. Following is the result of the above program.

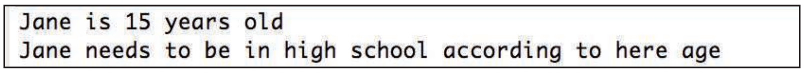


```
The right age for Middle School is: 10
The current age is less than high school age
```

However, we find that the if statement remains limited if the condition is found to be false. A false condition will mean that the next statement or the set of code in the brackets will be ignored. There are times where it is important to carry out specific executions according to the results of the Boolean expression placed in the parenthesis of the if statement. For such scenarios, we use the “if-else” combination. Below is a good syntax example for this control flow statement.

```
package java11.fundamentals.chapter11;
public class AgeCompare2 {
    public static void main(String[] args) {
        int age = 15;
        System.out.println(“Jane is ” + age + “ years old”);
        if (age < 14) {
            System.out.println(“She’s is too small to be in high school”);
        } else {
            System.out.println(“Jane needs to be in high school according to her age”);
        }
    }
}
```

The above code will produce the following result.



```
Jane is 15 years old
Jane needs to be in high school according to here age
```

This example shows that the failure of the if expression will execute the code block present with “else”. It is also possible to set up a long chain of “if-else-if” conditional statements. These statements can provide a series of actions, where they successfully follow a method of checking all possible conditions and produce the intended results.

It is better to stay away from this kind of conditional Java syntax as there is a great chance of producing logical errors that are hard to find even with the best IDE software tools for Java programming. We can use the “switch” statement when there can be multiple values and situations that need to be attended. Below is a program that shows how to use this method:

```
package java11.fundamentals.chapter11;
public class SwitchExample {
    public static void main(String[] args) {
        int numSides = 5;
        switch (numSides) {
        case 3:
            System.out.println(“It is a triangle”);
            break;
        case 4:
            System.out.println(“It is a quadrilateral”);
            break;
        case 5:
            System.out.println(“It is a pentagon”);
            break;
        case 6:
            System.out.println(“It is a hexagon”);
            break;
        default:
            System.out.println(“It is a polygon with more than six sides”);
        }
    }
}
```

The above program will produce the following result.

```
It is a pentagon
```

This program checks the number of sides of a shape to present the intended method. Clearly, there can be further cases that can easily be added in this switch code block. An advantage in this program is that the order can be arbitrary and therefore, it is easier to avoid mistakes or to find out if a particular condition is catered by the program.

11.10 | Loops

Loops are closely associated in programming languages with conditional operators. They control the program flow. Figure 11.4 shows the loops execution steps. We will discuss two types of loop statements that you can use in Java syntax – while loop and for loop.

11.10.1 While Loop

The "while" loop is by design an infinite loop. The expression presented in its parenthesis provides a Boolean answer true or false. This expression evaluation of Boolean value is perfect for ensuring that the loop can also work as the conditional operator.

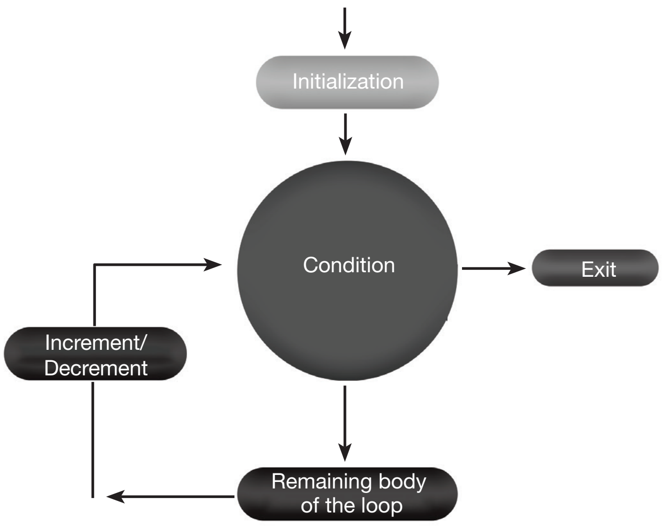
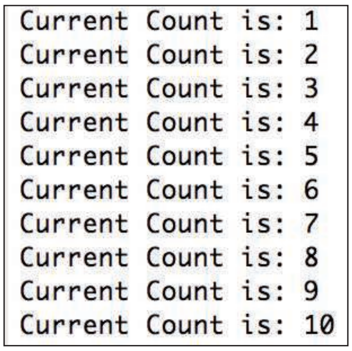


Figure 11.4 Loops execution steps.

There are two versions of the "while" loop. The first version is simple, where the expression is first tested and then the code block following the loop command is executed. The other version is the "do while" loop, in which it is possible to first run the code block and then test out the expression present in the while loop statement. Below is an example that describes loop sequence.

```
package java11.fundamentals.chapter11;
public class WhileExample {
    public static void main(String[] args) {
        int numCount = 1;
        while (numCount < 11) {
            System.out.println(“Current Count is: ” + numCount);
            numCount++;
        }
    }
}
```

The above program will print the value of numCount which initialized to 1 before entering into the while loop. The while loop has condition which checks if numCount is less than 11. If the condition is satisfied the program execution goes to the next line which prints the value of numCount. On the next line, the value of numCount increases by 1 and this way the loop continues until numCount is 10. Till this point, the program has already printed numbers from 1 to 10. After printing the numCount with value 10, the next line increases the numCount value by 1 which makes it 11. When the execution reaches to while, the expression check the condition and since the value of numCount is 11 which is not less than the comparison value 11, this makes the expression return false. Hence the execution exits the while loop.


```
Current Count is: 1
Current Count is: 2
Current Count is: 3
Current Count is: 4
Current Count is: 5
Current Count is: 6
Current Count is: 7
Current Count is: 8
Current Count is: 9
Current Count is: 10
```

Remember, it is always possible to set up an arbitrary condition in the loop, which will create an indefinite loop in the Java program. Using the “do” is a special case, which allows the program to at least execute the code block once, even if the “while” expression is found to be false in its first instance.

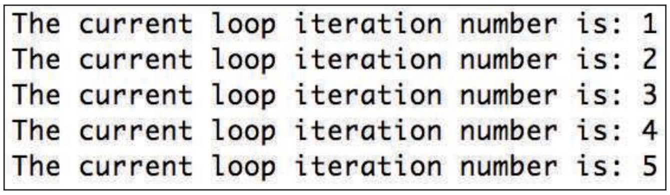
11.10.2 For Loop

The “for” loop statement is found in most legacy programming languages. This loop usually works with a counter and executes a block of code a finite number of times, which is specifically defined. It is an excellent method of reducing the overall code length, and ensures the possibility of several mathematical functions, which are required in several Java programs.

The “for” loop has a characteristic expression section in parenthesis. The first element initiates the counter variable, the second provides the loop termination condition, and the third element controls the number of iterations of the loop with an increment or a decrement. The code block that follows is the one which is executed in each loop iteration. Below is a simple example to help you learn its syntax.

```
package java11.fundamentals.chapter11;
public class LoopExample {
    public static void main(String[] args) {
        for (int count = 1; count < 6; count++) {
            System.out.println(“The current loop iteration number is: ” + count);
        }
    }
}
```

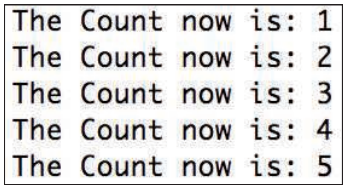
This program will run five times and present the iteration numbers from 1 to 5, each time carrying out a single increment. See the following output.


```
The current loop iteration number is: 1
The current loop iteration number is: 2
The current loop iteration number is: 3
The current loop iteration number is: 4
The current loop iteration number is: 5
```

However, Java also offers an enhanced way of running the count in the loop, which eliminates common problems that appear with the use of loops. Below is an example of this type of “for” loop.

```
package java11.fundamentals.chapter11;
public class BetterLoopExample {
    public static void main(String[] args) {
        int[] count = { 1, 2, 3, 4, 5 };
        for (int num : count) {
            System.out.println(“The Count now is: ” + num);
        }
    }
}
```

This program will also print the line five times, see the following output.



```
The Count now is: 1
The Count now is: 2
The Count now is: 3
The Count now is: 4
The Count now is: 5
```

This type of loop works by running the number of counts as many times as there are values in the employed integer array. This practice ensures that the control can have an independent layer, allowing programmers to stay away from simple mathematical and logical errors.

Flash ? Quiz Can you have a nested loop? (Hint: Nested loops are ones that contain another loop inside them.)

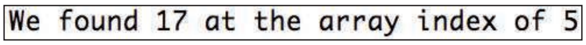
11.11 | Branching

It is possible to have improved control by using branching statements. Java offers two options for this function. This includes the use of “break” and “continue” statements. We have already presented an unlabeled form of break, which we can use to terminate any loop, as well as the different cases of switch, where this statement ends the execution of switch code block.

Let us take a look at a detailed example that shows the use of branching with other control flow elements.

```
package java11.fundamentals.chapter11;
public class DemoForBreak {
    public static void main(String[] args) {
        int[] intArray =
                { 68, 25, 74, 33, 95, 17, 53, 28, 1986, 53 };
        int reqNum = 17;
        int count;
        boolean found = false;
        for (count = 0; count < intArray.length; count++) {
            if (intArray[count] == reqNum) {
                found = true;
                break;
            }
        }
        if (found) {
            System.out.println(“We found ” + reqNum + “ at the array index of ” + count);
        } else {
            System.out.println(reqNum + “ is not stored in the array”);
        }
    }
}
```

This program is an excellent example of Java syntax structures which produces the following output.



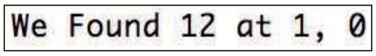
```
We found 17 at the array index of 5
```

The program initially describes an array with multiple values. We then create a counter that runs all these values one by one, to find if any value exactly matches the required number. The for loop is broken right then by the break command, showing an excellent use of the required branching. The ideal message can then be displayed by using another conditional statement. The program will iterate only if the value is continuously not found in the branched if statement.

Another way that “break” is employed is in a labeled form. In this form, it is designed to eliminate a specific case, which may be present as an external element in the program execution. Below is a program that explains this specific use of branching.

```
package java11.fundamentals.chapter11;
public class LabelBreakDemo {
    public static void main(String[] args) {
        int[][] arrayOne = {
            { 24, 87, 3, 18 },
            { 12, 76, 2000, 19 },
            { 22, 112, 67, 655 }
        };
        int lookFor = 12;
        int i;
        int j = 0;
        boolean numFound = false;
        search:
        for (i = 0; i < arrayOne.length; i++) {
            for (j = 0; j < arrayOne[i].length; j++) {
                if (arrayOne[i][j] == lookFor) {
                    numFound = true;
                    break search;
                }
            }
        }
        if (numFound) {
            System.out.println(“We Found ” + lookFor + “ at ” + i + “, ” + j);
        } else {
            System.out.println(lookFor + “ not in this array”);
        }
    }
}
```

According to the way that we have included 12 in this array, we can clearly see that it is in the second set of curly braces for the integer values. This describes that *i* will have the index of 1 in this position, while *j* will have the 0 position as 12 is the first element at the array position. This means that this program will print the following result.

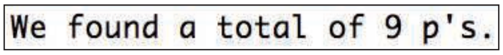


```
We Found 12 at 1, 0
```

This clearly describes that the loop did not run any longer when the loop value identified 12, as the break statement eliminated the entire “search” code block. To get more control, it is possible to employ another branching statement of “continue”. This is excellent for branching loops and calculating particular elements. This is evident in the following example.

```
package java11.fundamentals.chapter11;
public class DemoForContinue {
    public static void main(String[] args) {
        String searchP = "peter piper picked a peck of pickled peppers";
        int total = searchP.length();
        int numP = 0;
        for (int i = 0; i < total; i++) {
            // only want to count p's
            if (searchP.charAt(i) != 'p')
                continue;
            numP++;
        }
        System.out.println("We found a total of " + numP + " p's.");
    }
}
```

This program only counts character *p* when it is present, and simply moves back to increment *p* if the character is not found.

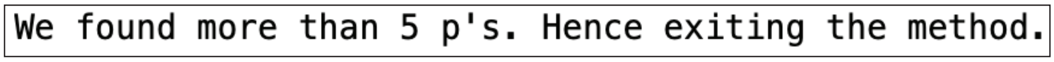


```
We found a total of 9 p's.
```

There is another method called as "return", which is used to stop a method and return the execution to the code, which follows the method invocation. See the following example which shows the use of "return". As soon as "return" is encountered the program execution exits the method.

```
package java11.fundamentals.chapter11;
public class DemoReturn {
    public static void main(String[] args) {
        String searchP = "peter piper picked a peck of pickled peppers";
        int total = searchP.length();
        int numP = 0;
        for (int i = 0; i < total; i++) {
            // only want to count p's
            if (searchP.charAt(i) != 'p')
                continue;
            numP++;
        }
        if(numP > 5) {
            System.out.println("We found more than 5 p's. Hence exiting the method.");
            return;
        }else {
            System.out.println("We found a total of " + numP + " p's.");
        }
    }
}
```

The above example is modified version of our earlier "continue" example. It contains a "if" condition which checks if numP is larger than 5 and if so it exits the method with the help of "return". See the following output of the above program.



```
We found more than 5 p's. Hence exiting the method.
```

Summary

We have described all the important elements that make up the Java syntax. These structures identify the presence of coding elements, blocks, and other facilities that are employed when writing code in Java. We also discussed specific options of Java such as the presence of static and non-static elements. The ability of the language to perform autoboxing and unboxing is also important. Java provides an excellent ability to develop program logic and carry out counting, arithmetic, and other related operations using a variety of code structures. We believe that this chapter will ensure that you have enough knowledge to become an efficient Java programmer. In this chapter, we have learned the following concepts:

1. Structure of a Java Program.
2. Identifiers, keywords, primitive classes, literals, variables, code blocks, and comments.
3. Static and non-static methods.
4. Enums and various Java operators.
5. Wrapper classes and how to use them.
6. Autoboxing and unboxing.
7. Expressions and loops and control flow.

In the Chapter 12, we will study the object-oriented programming world. We will cover abstraction, encapsulation, inheritance, and polymorphism in detail, accompanied by various examples, and overloading and overriding.

Multiple-Choice Questions

1. How many bits are in long?
   (a) 8
   (b) 12
   (c) 64
   (d) 20
2. How to start a block comment?
   (a) /
   (b) !-
   (c) *
   (d) /*
3. In Java, which of the following is not considered a numerical type?
   (a) int
   (b) char
   (c) float
   (d) short
   (e) double
4. The `System.in.read()` can be utilized for reading a character from keyboard.
   (a) True
   (b) False
5. In Java language, which one of the following is not an integer value?
   (a) 12
   (b) '23'
   (c) 10
   (d) 100

Review Questions

1. What is autoboxing and unboxing? Explain with examples.
2. What is the maximum number int data type can hold?
3. What is the use of enum?
4. What is literal? Give some examples.
5. Define a for each loop with an example.
6. What are the benefits of autoboxing and unboxing?
7. When should we use StringBuffer?
8. What is the use of instanceof operator?
9. Can you use Integer to hold long value?
10. Is StringBuilder immutable?
11. Can you pass int value to a method which accepts float?

Exercises

1. Write a program to add digits from 1 to 9 and repeat this process 10 times.
2. Write a program to create String array and display each element.
3. Write a program to use StringBuffer and StringBuilder. Print the time taken by each method and find out the fastest one. Explain why one is faster than the other.

Project Idea

Write a program to create a book library. Identify the classes you need to build. Create a String array to add book names and use this to display the required book information. Write a method to add a new book to this array and similarly add, remove, and update methods. Write a method that will print all the books in the library (from the String array).

Recommended Readings

1. Kathy Sierra and Bert Bates. 2018. *OCA Java SE 8 Programmer I Exam Guide*. McGraw Hill Education: New York
2. R. Nageswara Rao. 2016. *Core Java: An Integrated Approach*. Dreamtech Press: New Delhi